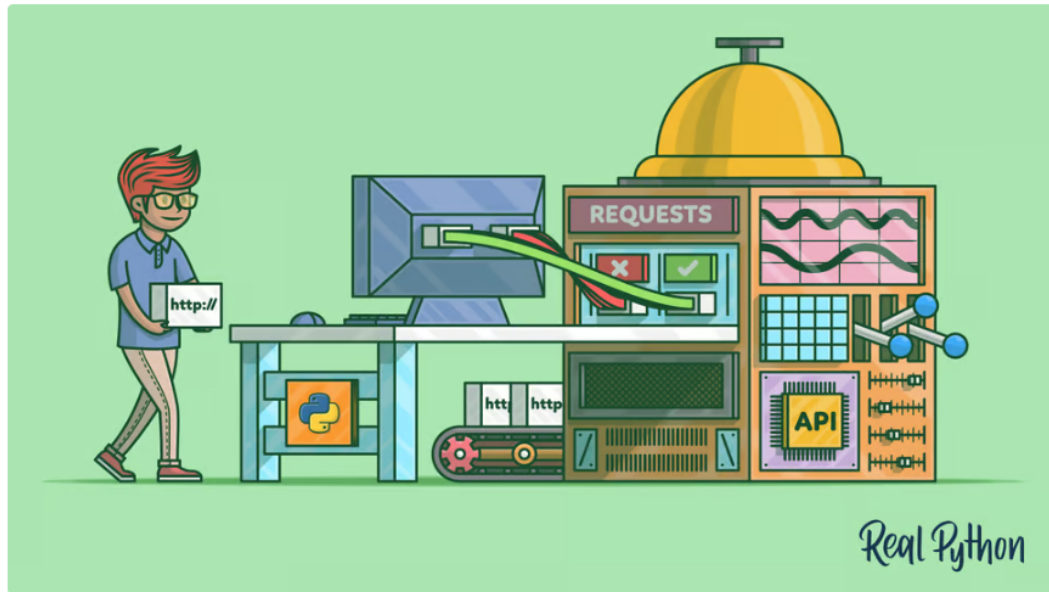


**Unisenac**  
Campus Pelotas




# Algoritmos e Estruturas de Dados I

CENTRO UNIVERSITÁRIO UNISENAC – CAMPUS PELOTAS  
CURSOS SUPERIORES: ESCOLA DE TECNOLOGIA  
PROF. EDÉCIO FERNANDO IEPSSEN



## Python's Requests Library (Guide)

by Alex Ronquillo · Feb 28, 2024 · 38 Comments

 intermediate  web-dev

A biblioteca [Requests](#) é o padrão de fato para fazer requisições HTTP em Python. Ela abstrai as complexidades de fazer requisições por trás de uma API bonita e simples para que você possa se concentrar em interagir com serviços e consumir dados em seu aplicativo.

Ao longo deste tutorial, você verá alguns dos recursos mais úteis que o Requests tem a oferecer, bem como maneiras de personalizar e otimizar esses recursos para diferentes situações que você pode encontrar. Você também aprenderá como usar o Requests de forma eficiente, bem como como evitar que solicitações a serviços externos deixem seu aplicativo lento.

### Neste tutorial, você aprenderá como:

- **Faça solicitações** usando os métodos HTTP mais comuns
- **Personalize** os cabeçalhos e dados das suas solicitações usando a sequência de consulta e o corpo da mensagem
- **Inspecione** dados de suas solicitações e respostas
- Faça solicitações **autenticadas**
- **Configure** suas solicitações para ajudar a evitar que seu aplicativo faça backup ou fique lento

Para a melhor experiência trabalhando neste tutorial, você deve ter [conhecimento geral básico de HTTP](#). Dito isso, você ainda pode conseguir acompanhar sem problemas sem ele.

# Introdução à biblioteca Requests do Python

Embora a biblioteca Requests seja um grampo comum para muitos desenvolvedores Python, ela não está incluída na [biblioteca padrão do Python](#) . Há [boas razões para essa decisão](#) , principalmente que a biblioteca pode continuar a evoluir mais livremente como um projeto autônomo.

**Nota:** Requests não suporta solicitações HTTP assíncronas diretamente. Se você precisa de suporte [assíncrono](#) em seu programa, você deve tentar [AIOHTTP](#) ou [HTTPX](#) . A última biblioteca é amplamente compatível com a sintaxe do Requests.

Como Requests é uma biblioteca de terceiros, você precisa instalá-la antes de poder usá-la em seu código. Como uma boa prática, você deve instalar pacotes externos em um [ambiente virtual](#) , mas pode escolher instalar `requests` em seu ambiente global se estiver planejando usá-lo em vários projetos.

Esteja você trabalhando em um ambiente virtual ou não, você precisará instalar `requests`:

Shell



```
$ python -m pip install requests
```

Uma vez [pip](#) terminada a instalação `requests`, você pode usá-lo em seu aplicativo. A importação `requests` parece com isso:

Python

```
import requests
```

# The GET Request

**Métodos HTTP**, como GET e POST, determinam qual ação você está tentando executar ao fazer uma solicitação HTTP. Além de GET e POST, há vários outros métodos comuns que você usará mais adiante neste tutorial.

Um dos métodos HTTP mais comuns é GET. O GET método indica que você está tentando obter ou recuperar dados de um recurso especificado. Para fazer uma GET solicitação usando Requests, você pode invocar `requests.get()`.

Para testar isso, você pode fazer uma GET solicitação à **API REST do GitHub** chamando `get()` com a seguinte URL:

Python

```
>>> import requests
>>> requests.get("https://api.github.com")
<Response [200]>
```

Parabéns! Você fez sua primeira solicitação. Agora você vai se aprofundar um pouco mais na resposta dessa solicitação.

## A resposta

A `Response` é um objeto poderoso para inspecionar os resultados da solicitação. Faça a mesma solicitação novamente, mas dessa vez armazene o valor de retorno em uma **variável** para que você possa ter uma visão mais detalhada de seus atributos e comportamentos:

Pitão



```
>>> import requests
>>> response = requests.get("https://api.github.com")
```

Neste exemplo, você capturou o valor de retorno de `get()`, que é uma instância de `Response`, e o armazenou em uma variável chamada `response`. Agora você pode usar `response` para ver muitas informações sobre os resultados da sua `GET` solicitação.

## Códigos de status

A primeira informação que você pode reunir `Response` é o código de status. Um código de status informa você sobre o status da solicitação.

Por exemplo, um `200 OK` status significa que sua solicitação foi bem-sucedida, enquanto um `404 NOT FOUND` status significa que o recurso que você estava procurando não foi encontrado. Há [muitos outros códigos de status possíveis](#) também para dar a você insights específicos sobre o que aconteceu com sua solicitação.

Ao acessar `.status_code`, você pode ver o código de status que o servidor retornou:

Pitão



```
>>> response.status_code  
200
```

`.status_code` retornou `200`, o que significa que sua solicitação foi bem-sucedida e o servidor respondeu com os dados que você estava solicitando.

Às vezes, você pode querer usar essas informações para tomar decisões em seu código:

Pitão

```
if response.status_code == 200:  
    print("Success!")  
elif response.status_code == 404:  
    print("Not Found.")
```



Com essa lógica, se o servidor retornar um 200 código de status, então seu programa imprimirá `. Success!` Se o resultado for um 404, então seu programa imprimirá `Not Found`.

Requests vai um passo além na simplificação desse processo para você. Se você usar uma `Response` instância em uma expressão condicional, então ela avaliará para `True` se o código de status for menor que 400, e `False` caso contrário.

Portanto, você pode simplificar o último exemplo reescrevendo a `if` declaração:

Pitão

```
if response:
    print("Success!")
else:
    raise Exception(f"Non-success status code: {response.status_code}")
```

No trecho de código acima, você verifica implicitamente se o `.status_code` de `response` está entre 200 and 399. Se não estiver, então você **levanta** uma **exceção** que inclui o código de status non-success em uma **f-string**.

**Nota:** Este teste de valor verdade é possível porque `.__bool__()` é um método sobrecarregado em `Response`. Isso significa que o comportamento padrão adaptado de `Response` leva o código de status em consideração ao determinar o valor verdade do objeto.

Tenha em mente que esse método *não* está verificando se o código de status é igual a 200. O motivo para isso é que outros códigos de status dentro do intervalo 200 to 399, como 204 `NO CONTENTE` 304 `NOT MODIFIED`, também são considerados bem-sucedidos no sentido de que fornecem alguma resposta viável.



# Content

A resposta de uma GETsolicitação geralmente tem algumas informações valiosas, conhecidas como `payload`, no corpo da mensagem. Usando os atributos e métodos de `Response`, você pode visualizar o payload em uma variedade de formatos diferentes.

Para ver o conteúdo da resposta em `bytes`, use `.content`:

Pitão



```
>>> import requests

>>> response = requests.get("https://api.github.com")
>>> response.content
b'{"current_user_url":"https://api.github.com/user", ...}'

>>> type(response.content)
<class 'bytes'>
```

Embora `.content` forneça acesso aos bytes brutos da carga útil da resposta, muitas vezes você desejará convertê-los em uma `string` usando uma `codificação de caracteres` como `UTF-8`. `response` fará isso para você quando acessar `.text`:

Pitão



```
>>> response.text
'{"current_user_url":"https://api.github.com/user", ...}'

>>> type(response.text)
<class 'str'>
```

Como a decodificação de bytes para a string requer um esquema de codificação, Requests tentará adivinhar a **codificação** com base nos **cabeçalhos** da resposta se você não especificar um. Você pode fornecer uma codificação explícita definindo `.encoding` antes de acessar `.text`:

Pitão



```
>>> response.encoding = "utf-8" # Optional: Requests infers this.
>>> response.text
'{"current_user_url": "https://api.github.com/user", ...}'
```

Se você der uma olhada na resposta, verá que é, na verdade, conteúdo JSON serializado. Para obter um dicionário, você pode pegar o `str` que você recuperou `.text` e desserializá-lo usando `json.loads()`. No entanto, uma maneira mais simples de realizar essa tarefa é usar `.json()`:

Pitão



```
>>> response.json()
{'current_user_url': 'https://api.github.com/user', ...}

>>> type(response.json())
<class 'dict'>
```

O `type` valor de retorno de `.json()` é um dicionário, então você pode acessar valores no objeto por chave:

# Cabeçalhos

Os cabeçalhos de resposta podem fornecer informações úteis, como o tipo de conteúdo da carga útil da resposta e um limite de tempo para armazenar a resposta em cache. Para visualizar esses cabeçalhos, acesse `.headers`:

Pitão

```
>>> import requests

>>> response = requests.get("https://api.github.com")
>>> response.headers
{'Server': 'GitHub.com',
 ...
 'X-GitHub-Request-Id': 'AE83:3F40:2151C46:438A840:65C38178'}
```

`.headers` retorna um objeto semelhante a um dicionário, permitindo que você acesse valores de cabeçalho por chave. Por exemplo, para ver o tipo de conteúdo do payload de resposta, você pode acessar `"Content-Type"`:

Pitão

```
>>> response.headers["Content-Type"]
'application/json; charset=utf-8'
```

Há algo especial sobre esse objeto de cabeçalhos semelhante a um dicionário, no entanto. A especificação HTTP define cabeçalhos como insensíveis a maiúsculas e minúsculas, o que significa que você pode acessar esses cabeçalhos sem se preocupar com suas maiúsculas:

Pitão

```
>>> response.headers["content-type"]
'application/json; charset=utf-8'
```

Quer você use a chave `"content-type"` ou `"Content-Type"`, você obterá o mesmo valor.

## Parâmetros da sequência de consulta

Uma maneira comum de personalizar uma GETsolicitação é passar valores por meio de parâmetros [de string de consulta](#) na URL. Para fazer isso usando `get()`, você passa dados para `params`. Por exemplo, você pode usar a API [de pesquisa de repositório](#) do GitHub para procurar repositórios Python populares:

Pitão

search\_popular\_repos.py

```
import requests

# Search GitHub's repositories for popular Python projects
response = requests.get(
    "https://api.github.com/search/repositories",
    params={"q": "language:python", "sort": "stars", "order": "desc"},
)

# Inspect some attributes of the first three repositories
json_response = response.json()
popular_repositories = json_response["items"]
for repo in popular_repositories[:3]:
    print(f"Name: {repo['name']}")
    print(f"Description: {repo['description']}")
    print(f"Stars: {repo['stargazers_count']}")
    print()
```

Ao passar um dicionário para o `params` parâmetro de `get()`, você pode modificar os resultados que retornam da API de pesquisa.

# Cabeçalhos de solicitação

Para personalizar cabeçalhos, você passa um dicionário de cabeçalhos HTTP para `get()` usar o `headers` parâmetro. Por exemplo, você pode alterar sua solicitação de pesquisa anterior para destacar termos de pesquisa correspondentes nos resultados especificando o `text-match` tipo de mídia no `Accept` cabeçalho:

```
Pitão text_matches.py

import requests

response = requests.get(
    "https://api.github.com/search/repositories",
    params={"q": "real python"},
    headers={"Accept": "application/vnd.github.text-match+json"},
)

# View the new `text-matches` list which provides information
# about your search term within the results
json_response = response.json()
first_repository = json_response["items"][0]
print(first_repository["text_matches"][0]["matches"])
```

O `Accept` cabeçalho informa ao servidor quais tipos de conteúdo seu aplicativo pode manipular. Nesse caso, como você espera que os termos de pesquisa correspondentes sejam destacados, você está usando o valor do cabeçalho `application/vnd.github.text-match+json`, que é um cabeçalho proprietário do GitHub Acceptem que o conteúdo é um formato JSON especial.

## Outros métodos HTTP

Além de GET, outros métodos HTTP populares incluem POST, PUT, DELETE, HEAD, PATCH, e OPTIONS. Para cada um desses métodos HTTP, Requests fornece uma função, com uma assinatura semelhante a `get()`.

**Nota:** Para testar esses métodos HTTP, você fará solicitações para [httpbin.org](https://httpbin.org). O serviço `httpbin` é um ótimo recurso criado pelo autor original de Requests, [Kenneth Reitz](#). O serviço aceita solicitações de teste e responde com dados sobre as solicitações.

Você notará que o Requests fornece uma interface intuitiva para todos os métodos HTTP mencionados:

Pitão

```
>>> import requests

>>> requests.get("https://httpbin.org/get")
<Response [200]>
>>> requests.post("https://httpbin.org/post", data={"key": "value"})
<Response [200]>
>>> requests.put("https://httpbin.org/put", data={"key": "value"})
<Response [200]>
>>> requests.delete("https://httpbin.org/delete")
<Response [200]>
>>> requests.head("https://httpbin.org/get")
<Response [200]>
>>> requests.patch("https://httpbin.org/patch", data={"key": "value"})
<Response [200]>
>>> requests.options("https://httpbin.org/get")
<Response [200]>
```

# O corpo da mensagem

De acordo com a especificação HTTP, POST, PUT, e as PATCH solicitações menos comuns passam seus dados pelo corpo da mensagem em vez de pelos parâmetros na string de consulta. Usando Requests, você passará o payload para o dataparámetro da função correspondente.

data pega um dicionário, uma lista de tuplas, bytes ou um objeto tipo arquivo. Você vai querer adaptar os dados que envia no corpo da sua solicitação às necessidades específicas do serviço com o qual você está interagindo.

Por exemplo, se o tipo de conteúdo da sua solicitação for `application/x-www-form-urlencoded`, você poderá enviar os dados do formulário como um dicionário:

Pitão



```
>>> import requests

>>> requests.post("https://httpbin.org/post", data={"key": "value"})
<Response [200]>
```

Você também pode enviar os mesmos dados como uma lista de tuplas:

Pitão



```
>>> requests.post("https://httpbin.org/post", data=[("key", "value")])
<Response [200]>
```



Se, no entanto, você precisar enviar dados JSON, então você pode usar o `json` parâmetro. Quando você passa dados JSON via `json`, Requests serializará seus dados e adicionará o `Content-Type` cabeçalho correto para você.

Como você aprendeu antes, o serviço `httpbin` aceita requisições de teste e responde com dados sobre as requisições. Por exemplo, você pode usá-lo para inspecionar uma `POST` requisição básica:

Pitão



```
>>> response = requests.post("https://httpbin.org/post", json={"key": "value"})
>>> json_response = response.json()
>>> json_response["data"]
'{"key": "value"}'
>>> json_response["headers"]["Content-Type"]
'application/json'
```

Você pode ver pela resposta que o servidor recebeu seus dados de solicitação e cabeçalhos conforme você os enviou. Requests também fornece essas informações a você na forma de um `PreparedRequest` que você inspecionará com mais detalhes na próxima seção.

# Solicitar inspeção

Quando você faz uma solicitação, a biblioteca Requests prepara a solicitação antes de realmente enviá-la ao servidor de destino. A preparação da solicitação inclui coisas como validar cabeçalhos e serializar conteúdo JSON.

Você pode visualizar o `PreparedRequest` objeto acessando `.request` em um `Response` objeto:

```
Pitão 

>>> import requests

>>> response = requests.post("https://httpbin.org/post", json={"key": "value"})

>>> response.request.headers["Content-Type"]
'application/json'
>>> response.request.url
'https://httpbin.org/post'
>>> response.request.body
b'{"key": "value"}'
```

A inspeção `PreparedRequest` dá acesso a todos os tipos de informações sobre a solicitação feita, como carga útil, URL, cabeçalhos, autenticação e muito mais.

Até agora, você fez muitos tipos diferentes de solicitações, mas todas elas tinham uma coisa em comum: eram solicitações não autenticadas para APIs públicas. Muitos serviços que você pode encontrar vão querer que você autentique de alguma forma.

# Autenticação

A autenticação ajuda um serviço a entender quem você é. Normalmente, você fornece suas credenciais a um servidor passando dados pelo `Authorization` cabeçalho ou um cabeçalho personalizado definido pelo serviço. Todas as funções de Requests que você viu até este ponto fornecem um parâmetro chamado `auth`, que permite que você passe suas credenciais:

```
Pitão
```

```
>>> import requests

>>> response = requests.get(
...     "https://httpbin.org/basic-auth/user/passwd",
...     auth=("user", "passwd")
... )

>>> response.status_code
200
>>> response.request.headers["Authorization"]
'Basic dXNlcjpwYXNzd2Q='
```

A solicitação será bem-sucedida se as credenciais que você passar na tupla `auth` forem válidas.

Quando você passa suas credenciais em uma tupla para o parâmetro, o Requests aplica as credenciais usando o [esquema de autenticação de acesso básico](#) `auth` do HTTP .

O esquema básico de autenticação

Mostrar/Ocultar

Você pode fazer a mesma solicitação passando credenciais de autenticação básica explícitas usando `HTTPBasicAuth`:

```
>>> from requests.auth import HTTPBasicAuth
>>> requests.get(
...     "https://httpbin.org/basic-auth/user/passwd",
...     auth=HTTPBasicAuth("user", "passwd")
... )
<Response [200]>
```

Embora você não precise ser explícito para autenticação Básica, você pode querer autenticar usando outro método. Requests fornece [outros métodos de autenticação](#) prontos para uso, como `HTTPDigestAuth` e `HTTPProxyAuth`.

Um exemplo real de uma API que requer autenticação é a API [de usuário autenticado](#) do GitHub . Este endpoint fornece informações sobre o perfil do usuário autenticado.

Se você tentar fazer uma solicitação sem credenciais, verá que o código de status é 401 Unauthorized:

```
>>> requests.get("https://api.github.com/user")
<Response [401]>
```

Se você não fornecer credenciais de autenticação ao acessar um serviço que as exige, você receberá um código de erro HTTP como resposta.

Para fazer uma solicitação à API de usuário autenticado do GitHub, primeiro você precisa gerar um [token de acesso pessoal](#) com o [escopo read:user](#). Então você pode passar esse token como o segundo elemento em uma tupla para `get()`:

```
Pitão

>>> import requests

>>> token = "<YOUR_GITHUB_PA_TOKEN>"
>>> response = requests.get(
...     "https://api.github.com/user",
...     auth=("", token)
... )
>>> response.status_code
200
```

Como você aprendeu anteriormente, essa abordagem passa as credenciais para `HTTPBasicAuth`, que espera um nome de usuário e uma senha e envia as credenciais como uma string codificada em Base64 com o prefixo "Basic ":

```
Pitão

>>> response.request.headers["Authorization"]
'Basic OmdocF92dkd...WpreM0SGRuUGY='
```

Isso funciona, mas não é a maneira correta de [autenticar com um token Bearer](#) — e usar uma entrada de string vazia para o nome de usuário supérfluo é estranho.

Com Requests, você pode fornecer seu próprio mecanismo de autenticação para consertar isso. Para tentar isso, crie uma subclasse de AuthBase e implemente `__call__()`:

```
Pitão custom_token_auth.py

from requests.auth import AuthBase

class TokenAuth(AuthBase):
    """Implements a token authentication scheme."""

    def __init__(self, token):
        self.token = token

    def __call__(self, request):
        """Attach an API token to the Authorization header."""
        request.headers["Authorization"] = f"Bearer {self.token}"
        return request
```

Aqui, seu mecanismo personalizado `TokenAuth` recebe um token e o inclui no `Authorization` cabeçalho da sua solicitação, definindo também o "Bearer " prefixo recomendado para a string.

Agora você pode usar esta autenticação de token personalizada para fazer sua chamada para a API de usuário autenticado do GitHub:

```
Pitão

>>> import requests
>>> from custom_token_auth import TokenAuth

>>> token = "<YOUR_GITHUB_PA_TOKEN>"
>>> response = requests.get(
...     "https://api.github.com/user",
...     auth=TokenAuth(token)
... )

>>> response.status_code
200
>>> response.request.headers["Authorization"]
```